The Use of a Microcomputer Based Array Processor for Real Time Laser Velocimeter Data Processing

James F. Meyers
NASA Langley Research Center
Hampton, VA

Fifth International Symposium on Applications of Laser Techniques to Fluid Mechanics
July 9-12, 1990
Lisbon, Portugal

The Use of a Microcomputer Based Array Processor For Real Time Laser Velocimeter Data Processing

James F. Meyers
NASA - Langley Research Center
Hampton, Virginia 23665 USA

ABSTRACT

The application of an array processor to laser velocimeter data processing is presented. The hardware is described along with the method of parallel programming required by the array processor. A portion of the data processing program is described in detail. The increase in computational speed of a microcomputer equipped with an array processor is illustrated by comparative testing with a minicomputer.

INTRODUCTION

In the beginning of laser velocimeter development, signal processing consisted of viewing the Doppler signal on a spectrum analyzer and writing down the measured frequency. In time more sophisticated signal processors were developed such as frequency trackers, high-speed burst counters and photon correlators which required computer data acquisition. In those days data was acquired with minicomputers then transferred to main frame computers for processing. By the mid 1970's the minicomputers had sufficient computing power to perform both the data acquisition and data processing tasks, if real time processing was not required. Unfortunately as computer capabilities and speeds increased so did the demands for more sophisticated data processing and real time presentation. These increased demands continued to keep the computer requirements at the minicomputer level. The development of personal computers in the mid eighties was an interesting curiosity and researchers found them very useful—for writing their research reports. By the end of the eighties, even though these small microcomputers had the computational capabilities of small minicomputers, they were only used by researchers restricted by small budgets. The present 80386 and 80486 machines now operate at the same speeds as minicomputers costing ten times as much. But, these *toy* computers are only single tasking and do not have the sophisticated operating system needed to acquire data and perform the statistical analysis.

In an another arena, researchers were defining computational tasks which taxed the limits of computer technology, even huge main frames. Researchers were finding that converted IBM-type business computers just were not fast enough for scientific calculations. This realization began the scientific computer industry with machines from the huge CDC cyber machines to the small minicomputers built by DEC, Hewlett-Packard, Data General, etc. Unfortunately researchers were developing problems faster than the computer companies could develop faster machines. At this point a few small companies, notably Floating Point Systems and CSPI, narrowed the task further: these computationally intensive scientific problems were basically simple floating point operations using standard mathematical computations. What if a specialized mathematical engine using parallel type programming were to be designed to run as an auxiliary unit on a main frame computer to perform these simple floating point operations quickly? By restricting the problem and using appropriate software techniques, circuits could be adjusted for very high speed operation. Soon these engines were migrating to smaller minicomputers and providing them with the computational capabilities of large main frame computers at a fraction of the price. The present paper will show that one or more of these strange engines can be placed in a *toy* computer to yield an extremely powerful machine for laser velocimeter data acquisition and real-time processing with statistical and graphical displays. The hardware components contained within the array processor are described. Example program code is described and compared with equivalent generic Fortran program steps.

What is an Array Processor?

The central processing unit (CPU), whether composed of discrete logic or a microprocessor, is a device for doing control operations and fundamental add, subtract, multiply and divide processes on integer data. Advanced operations such as floating point arithmetic or even advanced trigonometric operations rely on software to make the simple CPU perform these tasks. These software approximations require large amounts of code and thus long times to perform these fundamental operations. This has long been recognized as a major drawback in applying standard computer systems to scientific applications. Special hardware systems have been developed to perform these fundamental floating point operations in hardware. These systems are then appended to the CPU as peripheral processing units. Compilers were then written to take advantage of these units and transfer floating point operations to them reducing the required code while increasing computational speed. In microcomputers this processing unit, like the

CPU, is a single large scale integrated circuit such as an 80387 math coprocessor.

While math coprocessors such as the 80387 increase the computational capabilities and speed of the computer system, they are still general purpose units which can not be streamlined for maximum throughput. They are restricted to serial operation since they must compute transparently with standard language commands. This single operation mode is easily programmed, but extremely inefficient for the repetitive operations typical in scientific programming. Array processors break with this method of operation by performing their tasks in parallel using pipeline techniques to further increase throughput. The AT&T DSP32-C is a large scale integrated circuit specifically designed for digital signal processing applications. It consists of a 32-bit floating point multiplier, cascaded into a 40-bit arithmetic logic unit (ALU), cascaded into four 40-bit accumulators. The multiplier and ALU are in a pipeline and operate in parallel. Thus a multiply and an add or subtract can be performed simultaneously. This architecture makes the chip ideal for array processing applications. The AT&T chip can operate at speeds of up to 25 million floating point instructions per second (MFLOPs) yielding main frame computational speeds in a single chip.

Obviously an actual array processor consists of more than a single chip. The processor should also contain a high-speed integer math coprocessor to handle data transfer between the computer memory, the array processor memory, and the floating point unit. The integer coprocessor should also provide logical and integer arithmetic operations in support of the floating point processor and to complete the mathematical engine. The array processor also needs a bank of very high-speed static memory, DMA interface with controls to transfer data to/from the host computer, and the instruction decode and control hardware to make the unit operate efficiently. A block diagram of such an array processor is shown in Figure 1.

Parallel Programming

Parallel programming is not a new type of programming but a different way to solve a problem. One typically solves a problem by performing one operation after another until the answer is obtained, then moves on to the next set of data and repeats the operation. Parallel programming likewise performs one operation at a time, however it performs this operation on the entire set of data at once instead of one point at a time. It can also be thought of as vector or matrix arithmetic. Consider the following example: Develop the time history of an object's position as it

is dropped from a height of 100 meters with an initial downward velocity of 2 m/sec every 10 milliseconds for the first 4.0 seconds. The basic equation is thus:

$$s = 0.5at^2 + vt + s_o$$

where s is the present position, a is the acceleration of gravity (-9.8 m/s^2), t is time, v is the initial velocity (-2.0 m/s) and s_o is the initial height (100 m). Programming this function would require two arrays of $4/0.01+1$ or 401 elements for the position (s) and time (t) data. A Fortran version of the program would be:

```
1       s0  = 100.0
2       v   = -2.0
3       a   = -9.8
4       t(1) =  0.0
5       s(1) = 100.0
6       do i=2,401
7       t(i) = t(i-1) + 0.01
8       s(i) = 0.5*a*t(i)**2 + v*t(i) + s0
9       end do
```

The time consuming portion of this code segment lies in the loop, lines 6 through 9. Specifically line 8 requires 1 exponential, 3 multiplies, and 2 adds. Line 7 requires 1 add and 1 integer subtract. The time for memory fetches and puts, and the comparisons within the controlling do loop are neglected since their time is short compared to the mathematical operations. It should be noted however, memory operations to the computer's dynamic memory are much slower than operations to the array processor's static memory.

While the code, as written, is descriptively clear it is not very efficient, even for Fortran. The code should be rewritten as follows:

```
1       s0  = 100.0
2       v   = -2.0
3       a   = -9.8
4       t(1) =  0.0
5       s(1) = 100.0
5a      acc = 0.5*a
```

```
6       do i=2,401
7       t(i) = t(i-1) + 0.01
8       s(i) = (acc*t(i) + v)*t(i) + s0
9       end do
```

By using an additional variable (acc), 400 multiplies have been reduced to 1. Other operations are now missing or reduced: no exponentials, 2 multiplies instead of 3, and 2 adds. Thus readability in the code has been replaced by efficiency—but that is what comment lines are for. This efficient code can be further enhanced by using parallel programming with an array processor. After defining the original constants, the first task is to generate the time array using the ramp function to construct an array containing time in 10 millisecond increments from 0 seconds to 4.0 seconds (NOTE: The Fortran emulation is presented as commented code above the subroutine call):

```
1       s0  = 100.0
2       v   = -2.0
3       a   = -9.8
4       acc = 0.5*a

C       t(1)=0.0
C       do i=2,401
C       t(i) = t(i-1)+0.01
C       end do

5       call vframp(0.0,0.01,t,401)
```

Now that the time array is completed, the array processor's capability of performing an addition and a multiplication simultaneously can be used to perform the operation contained within the parentheses in line 8 above:

```
C       do i=1,401
C        s(i) = acc*t(i) + v
C       end do

6       call vsmsad(acc,t,v,s,401)
```

Again using the ability to simultaneously multiply and add, the resulting array (s) can be multiplied by the time array (t) and the offset (s0) added to complete the operation:

```
C           do i=1,401
C            s(i) = s(i)*t(i) + s0
C           end do

7           call vmsadd(s,t,s0,s,401)
```

Notice, as in Fortran, the results of an array processor operation can overwrite one of the input arrays, in this case the s array.

Obviously from the above example, emulation of the parallel processing code would actually be slower than the efficient Fortran code. However, the hardware looping and data management within the array processor along with the pipeline and optimized hardware multipliers and adders result in speed increases of several orders of magnitude. This example illustrates the difference between serial and parallel programming. Not all processes can be converted to parallel operation, but modern array processors are controlled by common, transparent subroutine calls allowing intermixing with standard serial Fortran operations.

Laser Velocimeter Data Processing Code

Once the array processor is initialized, memory allocated for arrays, and basic constants loaded, the processor is available for use within the data processing program. The beginning portion of the code will be shown to illustrate the type of operations that can be performed with the array processor and the ease of programming once the technique of parallel programming has been mastered.

The data acquired from the laser velocimeter signal processor usually consists of a packed integer word specifying the value of time for x cycles within the signal burst. For example, a high-speed burst counter output data word is made up of a 10- or 12-bit mantissa and a 4-bit exponent (power of 2) representing the number of master clock pulses occurring during the measurement period. The software must decouple these two segments of information from each measurement and convert them to a floating point number representing the measurement velocity. If the times between measurements are recorded by the data acquisition subsystem, they must also be converted to floating point numbers representing the length of these times. Assume that these two packed numbers have the following format:

measurement period - xxaaaaaaaaaabbbb

where

x - don't care

a - mantissa bits

b - exponent bits (power of 2)

and

interarrival time - eeaaaaaaaaaaaaaa

where

e - exponent bits (0.1 microseconds times 10 raised to the power ee)

a - mantissa bits

The following arrays are used:

rawu	- 16-bit integer array holding the input data from the high-speed burst counter
rawt	- 16-bit integer array holding the input interarrival time measurements
rawux	- 16-bit integer working array
rawtx	- 16-bit integer working array
u	- floating point array containing the velocity values
t	- floating point array containing the interarrival time values
scratch	- floating point scratch array

Begin the conversion process by masking and isolating the components of the velocity: Isolate the lowest 4-bits (b) and place them in array rawux where n is the number of measurements in the array as passed from the Fortran program:

```
call vsan16(#F,rawu,rawux,n)
```

Isolate the next 10- bits (a) and place them back in the original array:

```
call vsan16(#3FF0,rawu,rawu,n)
```

Logical shift right the mantissa bits by 4 to make the least significant bit (LSB) the 0 bit:

```
call vlsr16(rawu,4,rawu,n)
```

Convert the exponent to floating point and raise 2 to the exponent power:

```
call vflt16(rawux,scratch,n)
call vexp2(scratch,scratch,n)
```

Convert the mantissa to floating point and multiply by the exponential value to obtain the number of master clock pulses for each measurement:

```
call vflt16(rawu,u,n)
call vmul(u,scratch,scratch,n)
```

Load the value 1.0×10^{30} into the velocity array. Since n may be less than or equal to maxdata, this value will serve as an end-of-data indicator for the remaining portion of the program:

```
call vffill(a1e30,u,maxdata)
```

Divide the array into 32000 to convert the number of master clock pulses to measured signal frequency in MHz and place in the velocity array:

```
call vsdivr(a32k,scratch,u,n)
```

Use a similar code to convert the interarrival time data:

```
call vsan16(#C000,rawt,rawtx,n)
call vsan16(#3FFF,rawt,rawt,n)
call vlsr16(rawtx,14,rawtx,n)
call vflt16(rawtx,scratch,n)
call vssubl(s7,scratch,scratch,n)
call vexp10(scratch,scratch,n)
call vfl16u(rawt,t,n)
call vmul(t,scratch,scratch,n)
```

```
call vffill(a1e30,t,maxdata)
call vmov(scratch,t,n)
```

The frequency and interarrival time measurements are now stored in floating point notation in the arrays u and t respectively. The next step is to remove extraneous data, if any, from the ensemble, convert frequency to velocity and build the velocity histogram. This is accomplished by limiting the data to ±128 percent of the simple ensemble mean. The following code sequence continues processing and contains comments following the character !.

```
call meanv(u,s3,n)          !compute mean and store in s3
call vmul(s3,s5,s6,n1)      !multiply mean by 0.01 (in s5)
                            !and store result in s6
call vsmssl(s4,s6,s3,s5,n1) !multiply  0.01 of frequency
                            !(s6) by the value 128 (s4) and
                            !subtract from the mean (s3)
                            !and store result in s5
call vsmsad(s4,s6,s3,s6,n1) !multiply  0.01 of frequency
                            !(s6) by the value 128 (s4) and
                            !add to the mean (s3) and
                            !store result in s6
call vssubl(s2,s5,s5,n1)    !subtract the Bragg frequency
                            !from the lower limit of
                            !frequency
call vssubl(s2,s6,s6,n1)    !subtract the Bragg frequency
                            !from the upper limit of
                            !frequency
call vssubl(s2,u,u,n)       !subtract the Bragg frequency
                            !from the data in the velocity
                            !array
call vsmul(s1,s5,s5,n1)     !multiply lower limit of
                            !frequency by the fringe
                            !spacing to obtain velocity
```

```
call vsmul(s1,s6,s6,n1)          !multiply upper limit of
                                 !frequency by the fringe
                                 !spacing to obtain velocity
call vsmul(s1,u,u,n)             !multiply data in the velocity
                                 !array by the fringe spacing to
                                 !obtain velocity
call mnmxv(u,s1,s2,n)            !obtain the minimum (s1) and
                                 !maximum (s2) of the velocity
                                 !array
call vhistz(u,vhis,n256,s5,s6,n) !build a 256 bin histogram
                                 !of the velocity data (vhis)
                                 !limited by s5 and s6
call maxvi(vhis,s3,s4,n256)      !determine the maximum
                                 !number of occurrences in the
                                 !histogram and its index
```

The process continues in the manner described by Meyers (1988) using the array processor to complete the statistical calculations of the velocity data ensemble.

Performance Tests

The completed code was tested using a 12.5 MFLOP array processor placed in a 33 MHz 80386 microcomputer containing a 80387 math coprocessor. The coded high-speed burst counter data and the coded interarrival time data were placed on the hard drive (16 msec average access time). The test consisted of reading the coded data from disk, converting it to floating point, storing the floating point values to disk, performing the statistical analysis using the procedure to insure independent, non-biased data developed by Edwards and Meyers (1984) as implemented by Meyers (1988), and displaying the results in numeric form with plots of the three velocity histograms shown in Figure 2. The test data set consisted of three component velocity data with interarrival times for 49 measurement locations.

The previous minicomputer system used at Langley for laser velocimeter applications required 14 minutes, 40 seconds to perform the test without histogram presentation while the present

minicomputer only required 6 minutes, 50 seconds. The microcomputer system using the array processor required 2 minutes, 25 seconds to perform the test without histogram presentation and 4 minutes, 30 seconds with the histograms. Clearly the *toy* computer with its primitive operating system can perform, with the help of an array processor, the laser velocimeter data acquisition and processing tasks a factor of three faster than the present *real* computer with all of its sophistication.

Summary

The concept of array processing was presented and the hardware system for microcomputer applications described. The concept of parallel programming used by array processors was discussed using examples including the laser velocimeter data processing program. And finally performance tests using the system clearly show the major improvements in computing speed obtained using an array processor to solve computationally intensive problems.

Bibliography

Edwards, R.V. and Meyers, J.F. 1984 *An Overview of Particle Sampling Bias*, Second International Symposium on Applications of Laser Anemometry to Fluid Mechanics, Lisbon, Portugal.

Meyers, J.F. 1988 *Laser Velocimeter Data Acquisition and Real Time Processing Using a Microcomputer*, Fourth International Symposium on Applications of Laser Anemometry to Fluid Mechanics, Lisbon, Portugal.

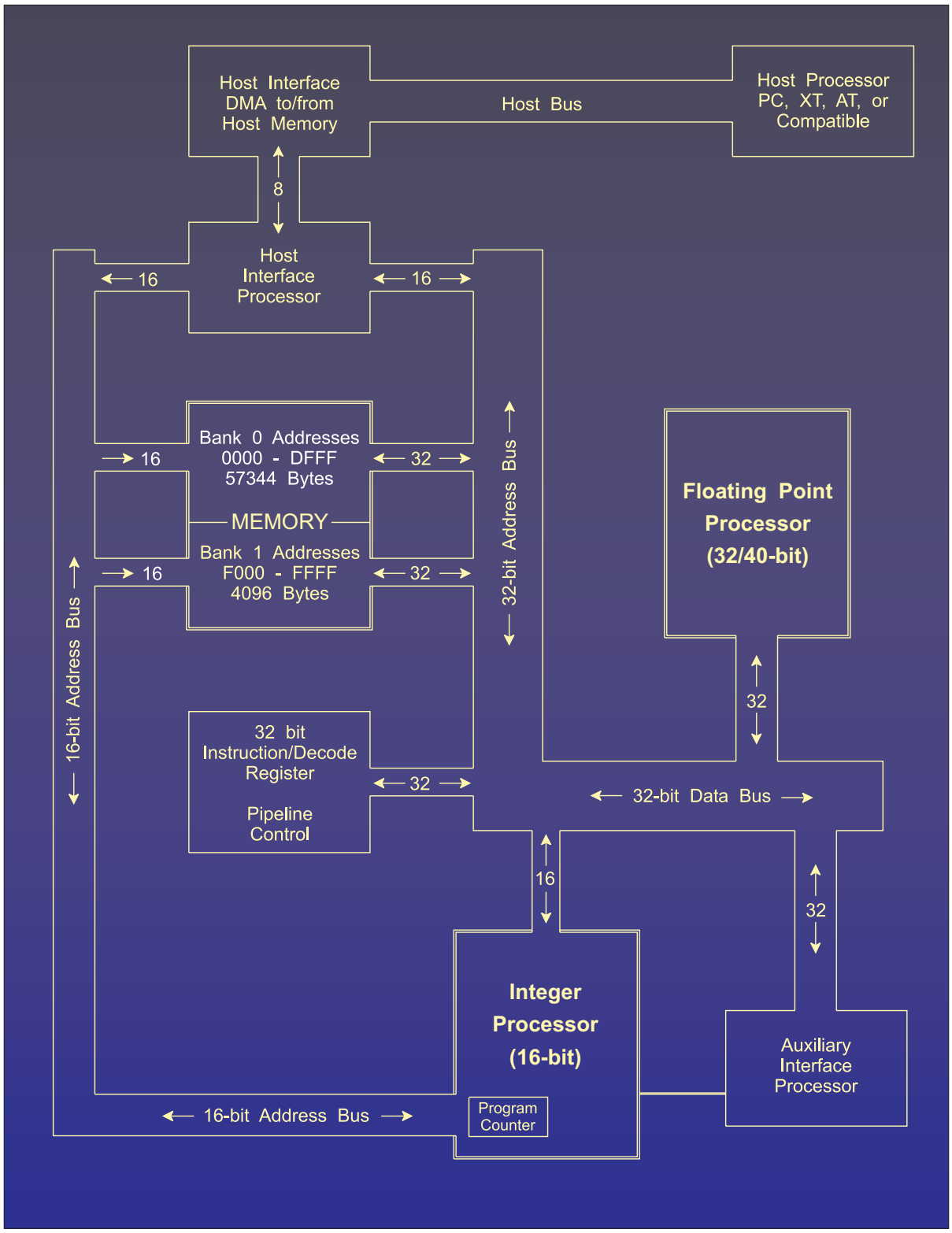

Figure 1.- Block diagram of a microcomputer based array processor.

Acquire Screen Optics LVABI Traverse Model

Run number: 201 9/17 3:55 Number of 4k blocks: 1

Sample Volume Position

Model Chord:	.0000	Streamwise	2.4350
Model Perpendicular:	.0000	Vertical	−3.9880
Model Span:	.0000	Cross Flow	5.2220

Component Statistics

	Channel 1	Channel 3	Channel 5
Mean velocity:	56.6699	15.7742	−13.8056
Standard deviation:	3.1240	1.5451	2.7219
Input data points:	962.0000	1442.0000	1837.0000
Integral time scale:	.0047	.0036	.0034
Minimum velocity:	−23.5267	−5.3051	−33.1469
Maximum velocity:	77.0190	22.9520	21.5880
% data accepted:	81.9421	99.0385	94.4959
Average data rate:	262.8526	331.6630	347.5351
V:R Correlation:	−.0315	−.0062	.0191

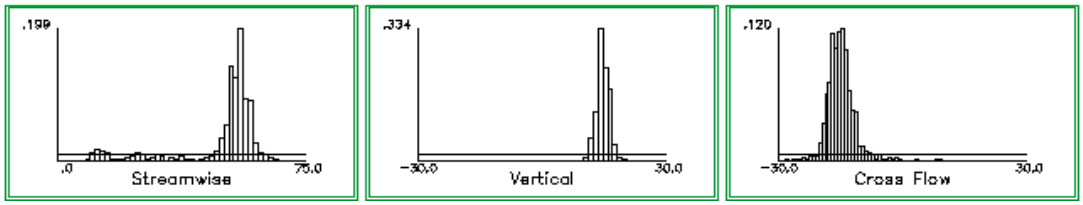

Figure 2.- Display screen from the laser velocimeter data acquisition and on-line processing program for a microcomputer.